Building a 3rd Generation Weather-Model System Test Suite

Paul Madden • Tom Henderson

paul.a.madden@noaa.gov

Definitions: Test Suite

- A collection of tests...
- ...that ensures against regression
- ...and gives a definitive pass/fail answer
- ...and automation
- ...and provides a framework.

Definitions: *System* Test Suite

- Unit tests (e.g. xUnit) – small chunks
- System tests – end-to-end
  - Compilers, MPI libraries, batch systems, task decomposition
- Evaluation
  - Run-vs-Baseline
  - Run-vs-Run

Definitions: Test-Suite Generations

- Gen 0: No tests. Manual tests. TLAR.
- Gen 1: Shell scripts
  - Provide some framework and automation
  - Grow by accretion/duplication, comprehensible only by a few experts
- Gen 2: Higher-level languages
  - Code re-use, modularity via OO design
  - Imperative style

Goals

- For the code under test...
  - Correctness
    - Run-vs-Baseline (ability to generate & use baseline)
    - Run-vs-Run
  - Breadth
    - Builds with different compilers, MPI libraries
    - Suite to provide Platform Interface
      - e.g how to interact with batch system

Goals

- For the test-suite users...
  - Easy to configure and run
  - Terse & verbose information in balance
  - Test-suite run time & coverage in balance
    - “Standard” and “Long” suites
    - Use threads for concurrency!
  - Fail early

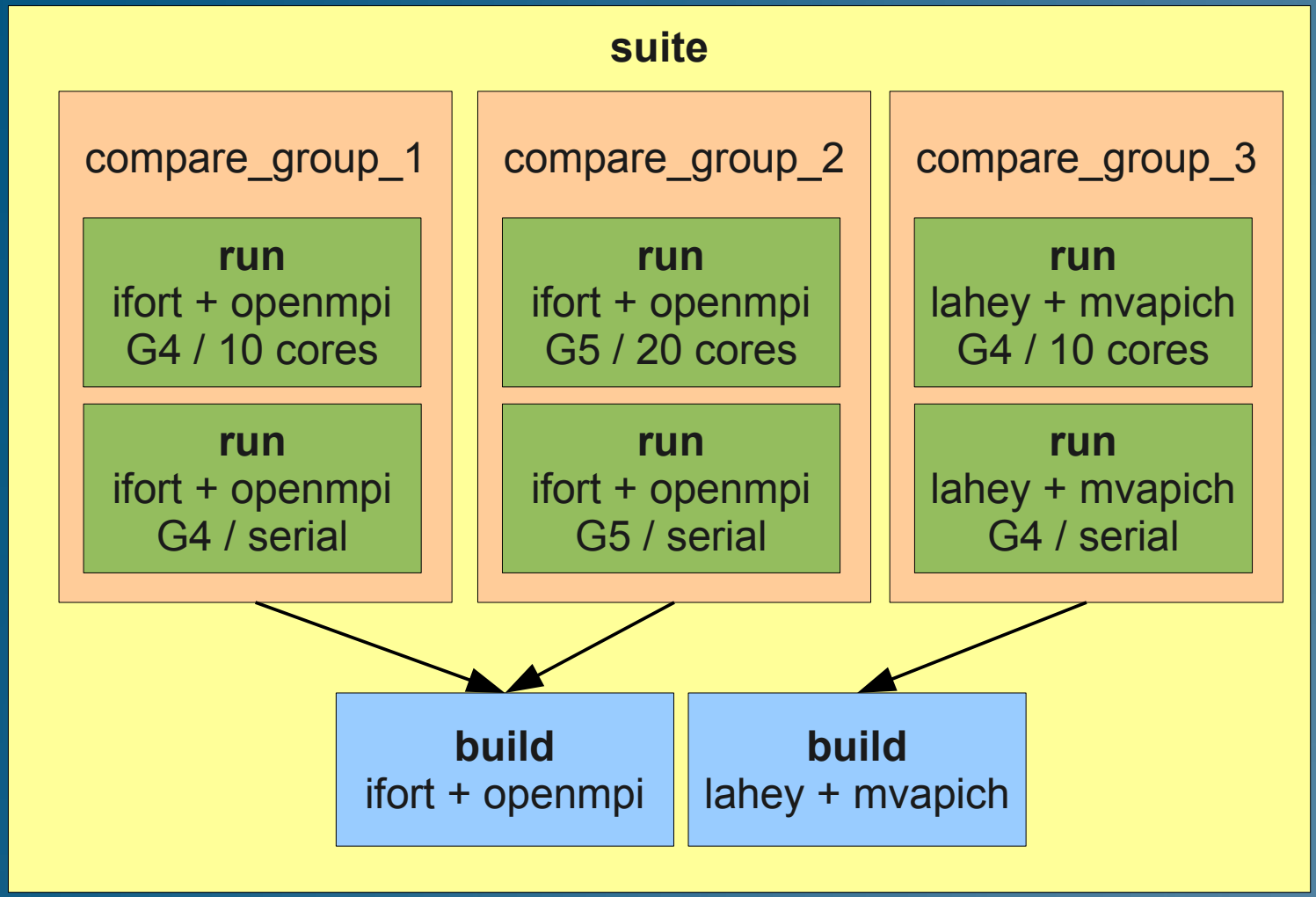
Goals

- For the test-suite developers...
  - Modularity for model and platform
    - Model Interface: how to build, how a run signals success, which output files to compare, etc.
  - Code re-use via libraries
  - Simplicity for easy support
  - Detailed logging for debugging

Design: Dependency-Driven

- Dependency-driven execution
  - Declarative vs imperative
  - Top level: define groups of comparable runs
    - Depends on: run definitions
  - Middle level: define runs
    - Depends on: build definitions
  - Bottom level: define builds
    - Depends on: external build automation system

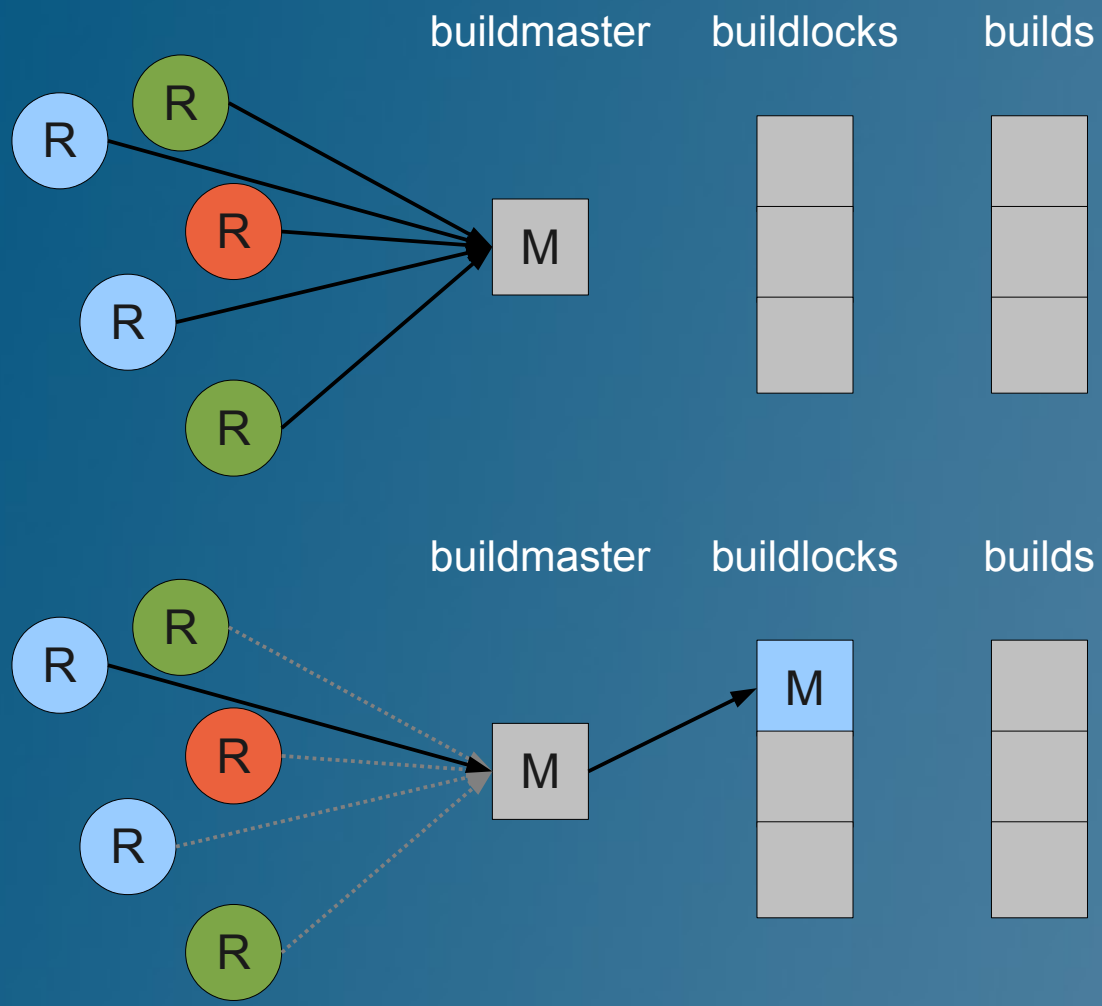
Design: Dependency-Driven



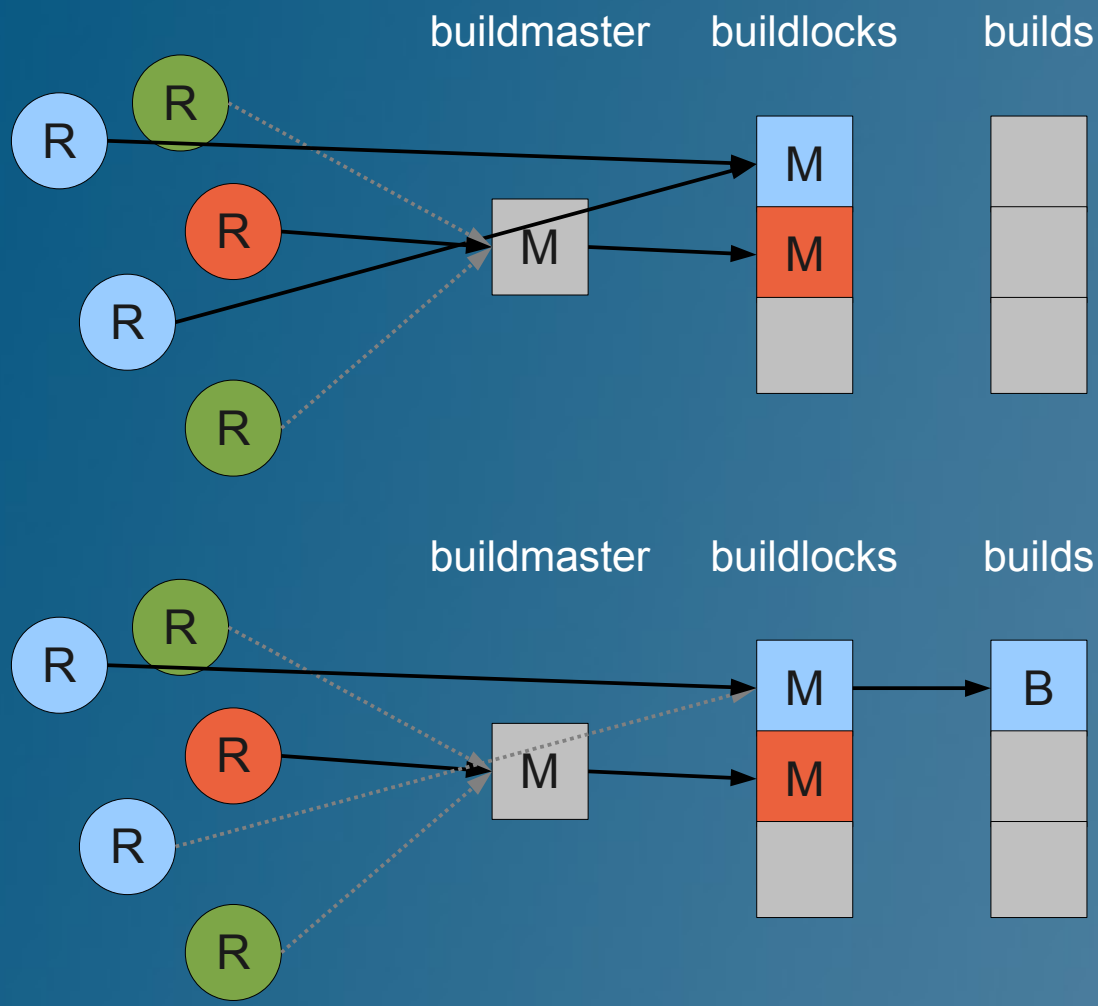
Design: Dependency-Driven

- Suites depend on compares
  - Compares tell suite: pass or fail
- Compares depend on runs
  - Runs tell compares: here's my output
- Runs depend on builds
  - Builds tell runs: here are executables
- If several runs need a build, let *one* of them build it while the others wait

Design: Dependency-Driven



Design: Dependency-Driven



Design: Dependency-Driven

- Benefits
  - No need to worry about order of operations
  - Nothing is built or run unless needed
  - No need for “if” conditionals in code
  - No combinatoric blow-up

Design: Composable Definitions

- Suite definition
  - “arch”: which batch system, etc. to use
  - “compare”: groups of comparable runs
  - Run names are *names of files* containing run definitions

```
arch: jet
compare:
  compare_group_1:
    - ifort_openmpi_4_10
    - ifort_openmpi_4_s
  compare_group_2:
    - ifort openmpi_5_20
    - ifort_openmpi_5_s
  compare_group_3:
    - lahey_mvapich_4_10
    - lahey_mvapich_4_s
```

Design: Composable Definitions

- Run definition
  - filename: *ifort_openmpi_4_10*
  - “baseline”: baseline snapshot to store or compare against
  - “build”: build to use
  - “namelists”: mods to apply to runtime Fortran namelist file

```
baseline: base_ifort_openmpi
build: ifort_openmpi
namelists:
  cntlnamelist:
    glvl: 4
    nz: 32
    physics: gfs
  queuenamelist:
    computetasks: 10
    maxqueuetime: 00:05:00
```

Design: Composable Definitions

- Build definition
  - filename: *ifort_openmpi*
  - Configuration options map onto external build system

```
arch: intel
mpi: openmpi
par: parallel
```

Design: Composable Definitions

create new file:
conf/runs/ifort_openmpi_4_20

```
extends: ifort_openmpi_4_10
namelists:
  queuenamelist:
    computetasks: 20
```

modify suite file:
conf/suites/standard

```
arch: jet
compare:
  compare_group_1:
    - ifort_openmpi_4_20
    - ifort_openmpi_4_10
    - ifort_openmpi_4_s
  compare_group_2:
    - ifort openmpi_5_20
    - ifort_openmpi_5_s
  compare_group_3:
    - lahey_mvapich_4_10
    - lahey_mvapich_4_s
```

Design: Composable Definitions

conf/runs/intel_cpu_gfs_10

```
extends: intel_cpu_gfs
build: intel_cpu_p
namelists:
  queuenamelist:
    computetasks: “10”
    maxqueuetime: “00:05:00”
```

conf/runs/intel_cpu_gfs

```
baseline: intel_cpu_gfs
namelists:
  cntlnamelist:
    glvl: 5
    nz: 32
    physics: 'gfs'
```

conf/builds/intel_cpu_p

```
arch: intel
hw: cpu
par: parallel
```

Design: Composable Definitions

```
$ nimts show run intel_cpu_gfs_10

conf/builds/intel_cpu_p
  arch: intel
  hw: cpu
  par: parallel
conf/runs/intel_cpu_gfs_10
  baseline: intel_cpu_gfs
  build: intel_cpu_p
  extends: intel_cpu_gfs
  namelists:
    cntlnamelist:
      glvl: 5
      nz: 32
      physics: gfs
    queuenamelist:
      computetasks: 10
      maxqueuetime: 00:05:00
```

Design: Comparisons

- Run-vs-run handled via suite definition
- Run-vs-baseline
  - `nimts baseline` produces “baseline” directory
  - Run definition defines which baseline the run should read/write
  - Runs compete via mutex system to contribute their output to baseline for their group
  - Presence of a “baseline” directory implies baseline comparison

Design: Terse vs Verbose

- Immediate Logger
  - Messages appear on console + in log file
  - Output from different threads may be interspersed
- Delayed Logger
  - Messages go only to log file
  - Extremely verbose e.g. build output
  - Delayed logger messages collected & flushed, access to log file controled by mutex

Design: Multithreaded for Speed

- One thread per compare group / run / build
  - Concurrency derived from dependencies
- Each task proceeds when dependencies are satisfied
  - e.g. Comparisons between runs in one compare group happen as soon as *those* runs complete
  - Allows early-as-possible failure
- Front-end / compute-node work split

Design: Convention Over Configuration

- Baselines
  - Simple presence of “baseline” directory (real or symlink) implies “compare against baseline”
- Configuration
  - Build definitions in conf/builds
    - Any filename here can be referred to in a run definition
  - Run definitions in conf/runs
    - Any filename here can be referred to in a suite definition

Design: Portability

- 6 methods define Platform Interface
  - Primarily batch-system issues (how to queue, monitor, delete jobs, etc.)
- 8 methods define Model Interface
  - How to prepare an isolated build, syntax of build command, how to check if a model run completed, which output files to compare or store
- Compile/link details left to model's build system

Design: Portability

Model 1

```
def arch_build_pre(buildspec)
  # no-op
end
```

Model 2

```
def arch_build_pre(buildspec)
  build=buildspec['build'].squeeze
  buildbase=build.sub(/\s*serial\s*/,'')
  builddir=build.sub(' ','_')
  dstdir=buildspec['buildroot']+'/'+builddir
  FileUtils.mkdir(dstdir)
  logd "Made directory: #{dstdir}"
  n='FIMsrc'
  src=valid_dir(File.expand_path('../../'+n))
  dst=dstdir+'/'+n
  FileUtils.cp_r(src,dst)
  logd "Copied #{src} to #{dst}"
  buildspec['buildsrc']=valid_dir(dst)
  logd "Set build source directory: #{dst}"
  n='FIMrun'
  src=valid_dir(File.expand_path('../../'+n))
  dst=dstdir+'/'+n
  FileUtils.mkdir(dst)
  Dir.glob(src+'/*') do |e|
    FileUtils.cp(e,dst) unless File.directory?(e)
  end
  logd "Copied #{src}/* to #{dst}"
  buildspec['buildrun']=valid_dir(dst)
  logd "Set build run directory: #{dst}"
end
```

Implementation

- Driver code in Ruby
  - Good maintenance & extension experiences
  - Dynamic dispatch, e.g. command-line arguments translated directly to method calls
  - Good libraries like
    - Logger: immediate and delayed logs
    - Thread: multithreading & mutexes
    - YAML: config files
    - MD5: test-suite data set verification
    - Fortran namelist handler (custom)

Experiences So Far

- Testing NIM model on two supercomputers
- Adapted test suite to FIM model for continuous integration tests on new system
- Developers already modifying their own test suites
- Re-used some components for non-model test suite
- Goals met

Thanks.